Faculdade de Engenharia da Universidade do Porto

FEUP

Manual do SimTwo

(v0.1)

Paulo Gomes Costa (paco@fe.up.pt)

A. Paulo Moreira (amoreira@fe.up.pt)

Hugo Pinto Alves (hugo.alves@fe.up.pt)

Filipe Neves dos Santos (fbnsantos@fbnsantos.com)

Maio de 2011

Índice

1. Introdução 3

1.1. Funcionalidades do Simulador 4

2. Biblioteca de Simulação de Corpos Rígidos 9

2.1. Simulação dinâmica de corpos rígidos 9

2.2. Simulação de Colisões 11

3. Configuração do simulador 12

4. Modos de Controlo 16

4.1. Controlo dos robôs via *Script* 16

4.2. Controlo dos robôs via Rede 17

4.3. Controlo dos robôs por Porta Série 17

Referências 18

Anexo 19

Tipos disponíveis no Script 19

Funções disponíveis no Script 20

1. Introdução

A plataforma de simulação *SimTwo* é um sistema de simulação *3D* realista onde é possível implementar vários tipos de robôs, dos quais se destacam os robôs móveis, manipuladores industriais, humanóides e células de manufactura. Os corpos rígidos representados no ambiente de simulação possuem um conjunto de propriedades físicas que permitem configurar a sua estrutura mecânica e simular o comportamento dinâmico dos objectos reais.

Esta ferramenta foi desenvolvida pelo autor Paulo Costa e pode ser encontrada em [1].

Na figura 1, apresenta-se a interface de visualização *3D* onde é possível observar o comportamento do sistema em tempo real.

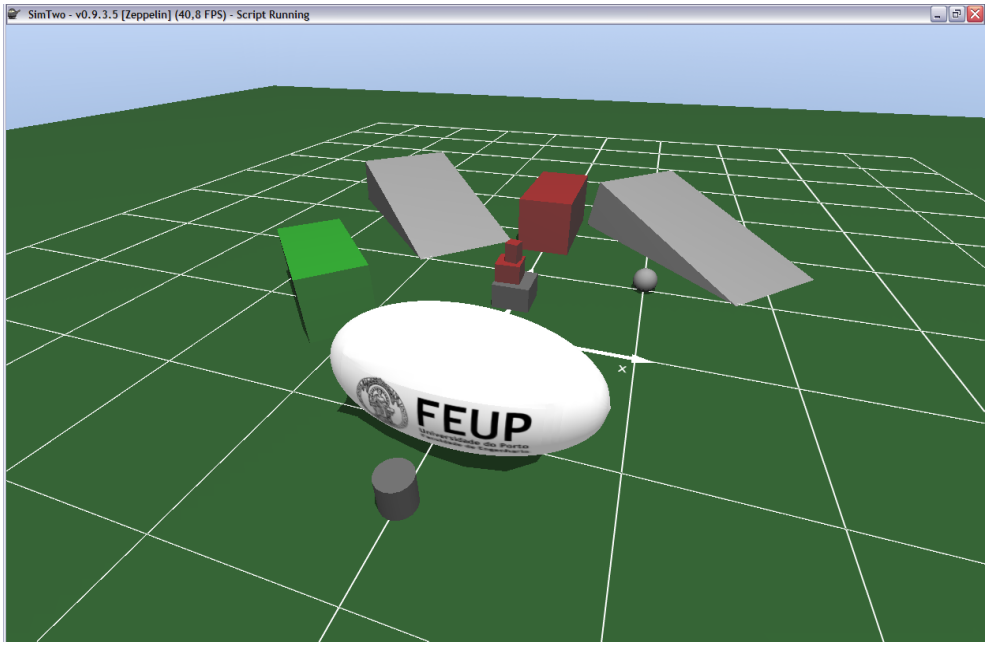


Figura 1 – Janela de visualização da interface gráfica *3D* do *SimTwo*

O realismo retratado neste simulador, traduz-se na introdução da dinâmica real de um robô, e permite decompor o mesmo em duas partes:

- Realismo introduzido nos sistemas de corpos rígidos, utilizando a biblioteca *ODE* (*Open Dynamics Engine*);
- Realismo aplicado nos modelos dos motores eléctricos e controladores.

A biblioteca *ODE* é direccionada para a simulação em tempo real, pelo que é dada prioridade à rapidez e estabilidade face à precisão. Esta é vocacionada para a simulação de estruturas articuladas, tais como veículos terrestres ou criaturas com pernas. Os robôs enquadram-se normalmente neste grupo. As estruturas articuladas, ou rodas, nos robôs encontram-se associadas a eixos que normalmente supõem um sistema de accionamento.

A modelização dos modelos dos motores tem como objectivo capturar as não linearidades inerentes a estes actuadores, tais como limitações de corrente, saturação da tensão aplicada e atrito de *Coulomb*. Estas não linearidades são factores imutáveis nos motores de todos os robôs.

Para permitir simular o movimento dos corpos rígidos, a plataforma de simulação disponibiliza um sistema de accionamento baseado em motores de corrente contínua e controlador[1], cujos seus parâmetros podem ser ajustados.

O controlo do robô pode ser efectuado no simulador através de uma linguagem de *Script*, editável e compilável no próprio simulador, ou então, através de uma aplicação remota que comunica com o *SimTwo* por *UDP* ou porta série.

1.1. Funcionalidades do Simulador

Como foi referido anteriormente o *SimTwo* é dotado de uma janela de visualização onde é possível observar o comportamento do mundo a simular. Esta janela permite:

- Fazer zoom com o scroll do rato;

[1] PID de referência de posição ou velocidade, ou realimentação de estado.

- Mudar o ângulo de visualização carregando em *Ctrl* e na tecla esquerda do rato, movendo-o ao mesmo tempo;
- Pegar em objectos (bola, robôs, etc), movê-los para cima, baixo, esquerda e direita, usando o rato (para os afastar ou aproximar deve-se premir a tecla *q* ou *a*, respectivamente, ao mesmo tempo que se carrega no rato);
- Trocar ou abrir outras janelas do SimTwo (*scene* – configuração do ambiente a simular, editor de *script*, janela de configuração, folha de cálculo, janela de gráficos), trocar de cenário de simulação, captar uma imagem em tempo real do simulador, abortar o simulador, premindo a tecla direita do rato.

Para além da janela de visualização gráfica *3D*, o *SimTwo* fornece também uma janela de configuração da plataforma de simulação, figura 2. Através desta é possível configurar parâmetros gráficos da interface gráfica *3D*, figura 2 a), parâmetros da configuração do robô, figura 2 b), bem como parâmetros da configuração física dos objectos, figura 2 c). De seguida são descritas algumas tarefas básicas desta janela de configuração:

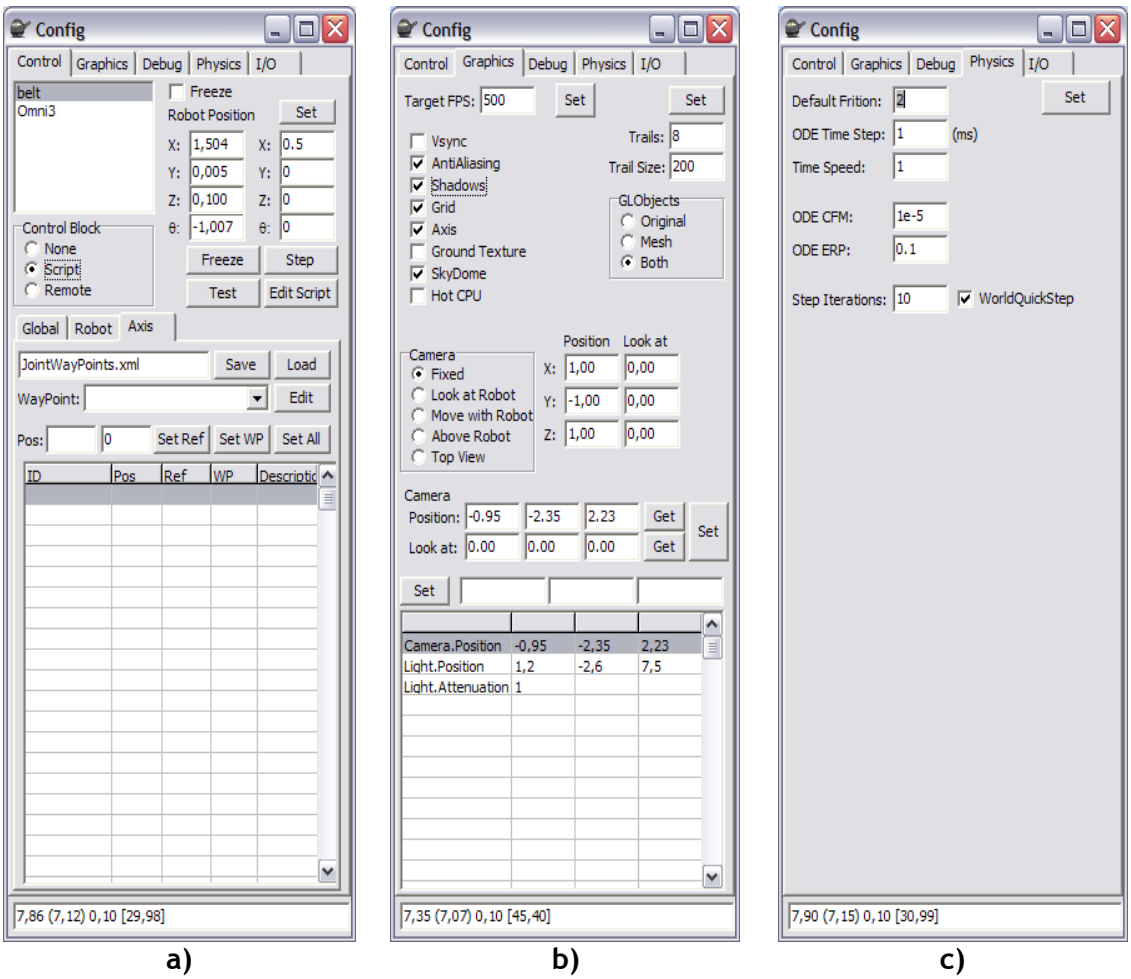
- Congelar a simulação;
- Alterar a posição do robô (seleccionando-o na lista de robôs);
- Definir o tipo de controlo (none, script ou remote);
- Visualizar as características do robô (medidas dos sensores, tensões aplicadas aos motores, etc);
- Editar os waypoints e introduzir referências instantâneas a cada articulação;
- Alterar e definir características gráficas de visualização (posição da câmara, etc);
- Fazer debug das propriedades de cada motor e controlador definido para cada articulação;
- Alterar características físicas dos objectos inerentes à biblioteca *ODE*;
- Definir o porto de comunicação por *UDP*;

- Iniciar/Terminar a comunicação por porta-série com um dispositivo externo e enviar/ receber mensagens;

O simulador é também dotado de um editor de código que permite a execução de código escrito numa variante da linguagem *Pascal* que é um subconjunto do *ObjectPascal* (tal como existe para o *Delphi* e *Free Pascal*), figura 3.

Na figura 4, ilustra-se a janela de gráficos. Esta janela permite registar a evolução de algumas variáveis em função do tempo e também visualizar esta progressão em tempo real.

Como foi referido anteriormente, o *SimTwo* permite simular diferentes configurações de robôs. Para tal, este simulador recorre a ficheiros com uma sintaxe *XML* editáveis a partir de uma janela fornecida pela plataforma de simulação, figura 5.



a) **b)** **c)**

Figura 2 – Janela de configuração do *SimTwo*

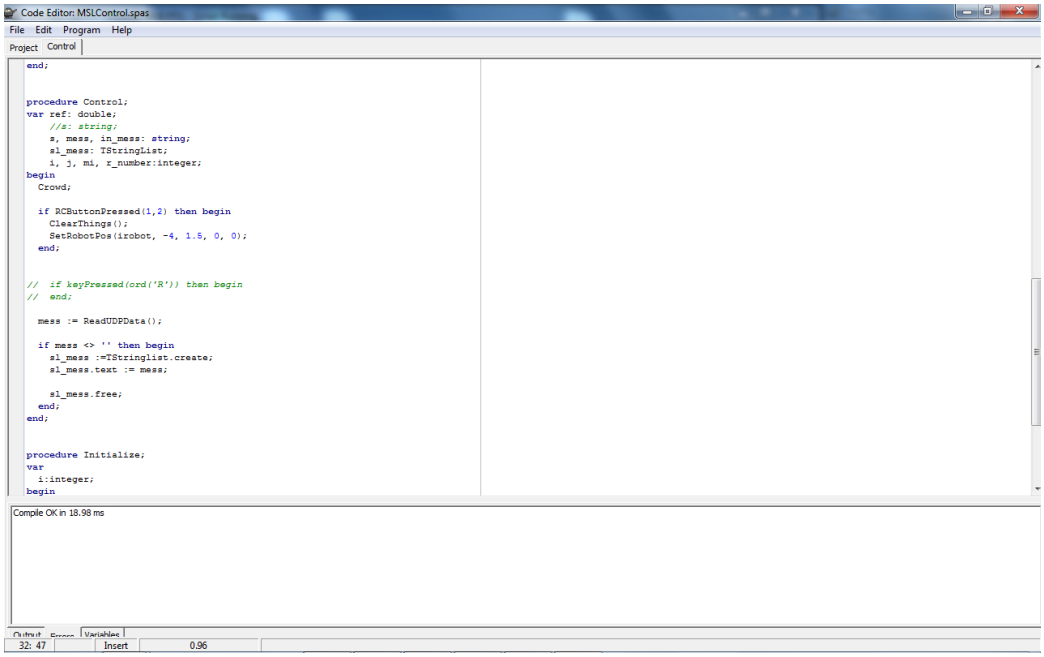


Figura 3 – Janela do editor de código do *SimTwo*

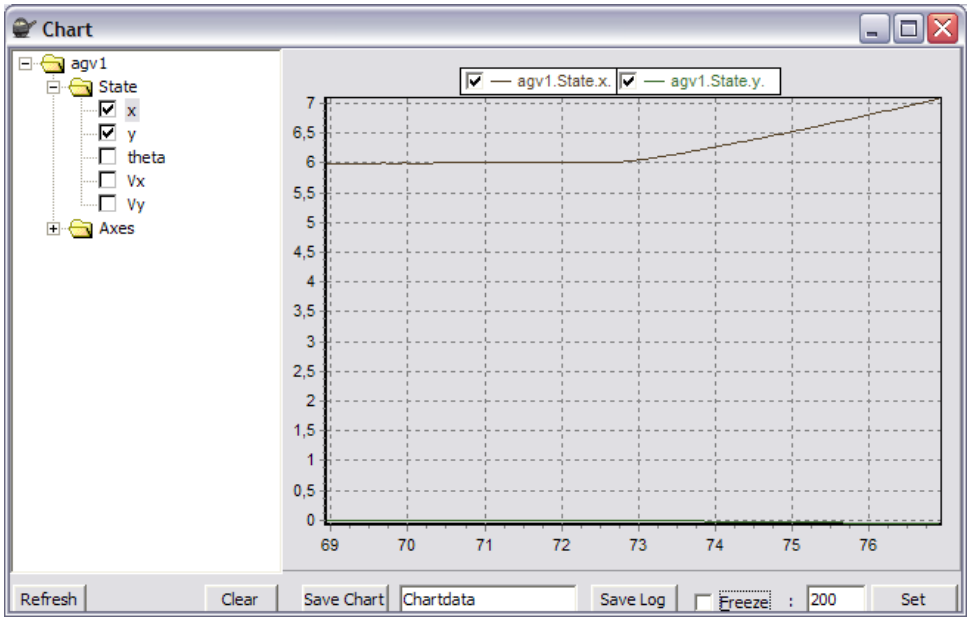


Figura 4 – Janela de gráficos do *SimTwo*

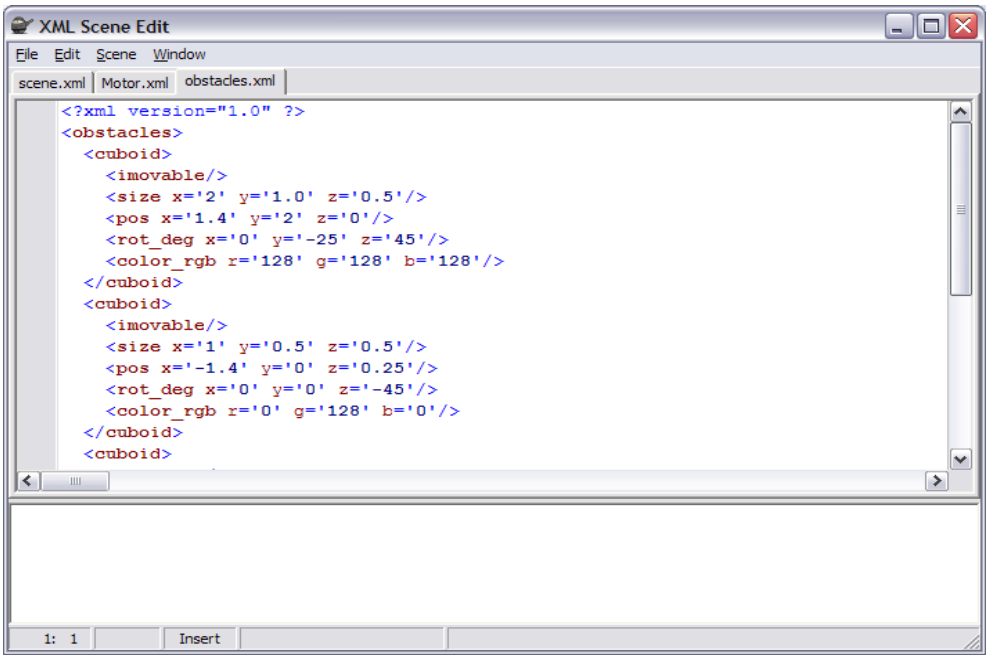


Figura 5 – Janela de edição em *XML* dos objectos do *SimTwo*

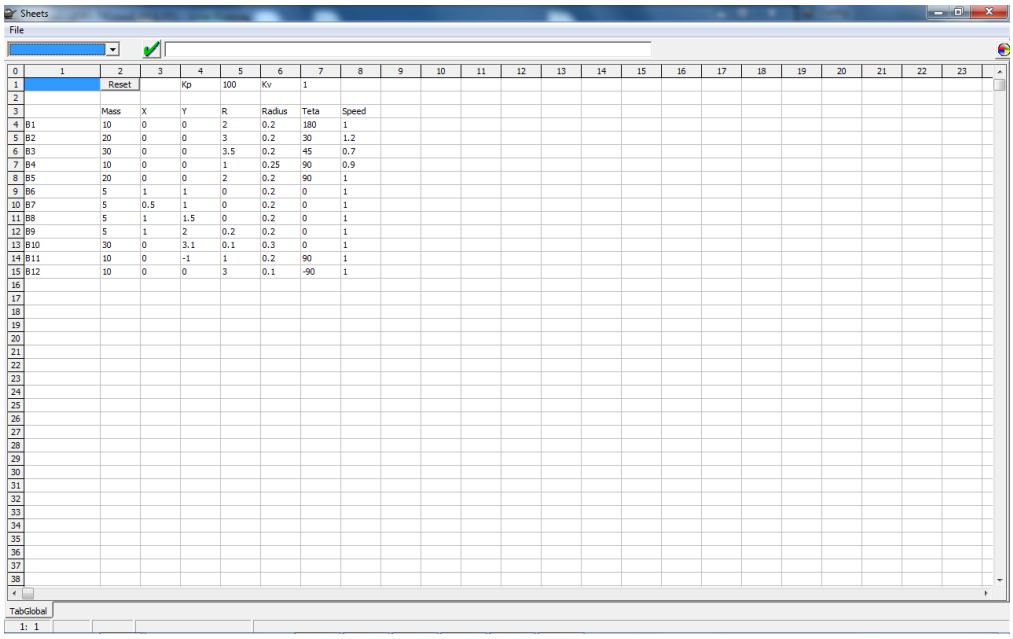


Figura 6 – Janela de edição de dados, tipo folha de cálculo, do *SimTwo*

2. Biblioteca de Simulação de Corpos Rígidos

O *SimTwo* usa a biblioteca *ODE* para a simulação de corpos rígidos.

A biblioteca supõe dois níveis de simulação: o nível da dinâmica que se encarrega das propriedades dinâmicas dos corpos, tais como massa e velocidade; e o nível das colisões, que se ocupa da forma dos corpos. Estes níveis são propositadamente separados, com o intuito de maximizar a flexibilidade e utilidade da biblioteca. Desta forma é possível utilizar uma biblioteca de simulação de colisões externa juntamente com a biblioteca interna de simulação da dinâmica, ou vice-versa.

Relativamente ao nível da simulação dinâmica, os corpos são desprovidos de forma, sendo representados pela sua massa e momentos de inércia.

2.1. Simulação dinâmica de corpos rígidos

A simulação da dinâmica de corpos rígidos envolve dois tipos de entidades: *body* e *joint*. As entidades *body* representam os corpos rígidos, e correspondem a distribuições arbitrárias de massa, representadas pela sua massa e pela matriz 3 x 3 do seus momentos de inércia. Sobre os corpos são exercidos binários e forças, que podem ser aplicados directamente ou por intermédio de junções com outros corpos. As entidades *joint* correspondem a junções entre os corpos, permitindo a transmissão de forças e binários entre estes. As junções impõem restrições ao movimento, sendo que a cada tipo de junção correspondem determinadas restrições.

Os vários tipos de junções suportados pela biblioteca são descritos na tabela 1. Para mais detalhes, consultar a documentação da biblioteca em [2].

Na figura 7 é ilustrado o princípio conceptual de funcionamento de uma junção do tipo hinge,cujas restrições são semelhantes às apresentadas por

uma dobradiça. Como se pode perceber, esta junção permite apenas o movimento angular relativamente a um eixo, mas previne qualquer tipo de movimento linear.

Tabela 1 – Tipos de junções suportados pela biblioteca *ODE*

Tipo	Descrição	Graus de Liberdade
Ball	Junção de rótula	3 angulares
Hinge	Junção do tipo dobradiça	1 angular
Slider	Junção de deslizamento linear	1 linear
Universal	Junção do tipo *cardan*	2 angulares
Hinge2	Junção de dobradiças em série	2 angulares
Fixed	União fixa entre corpos	nenhum
Contact	Junção de contacto	(especial)
Amotor	Junção que permite controlar as velocidades angulares relativas entre dois corpos	(especial)

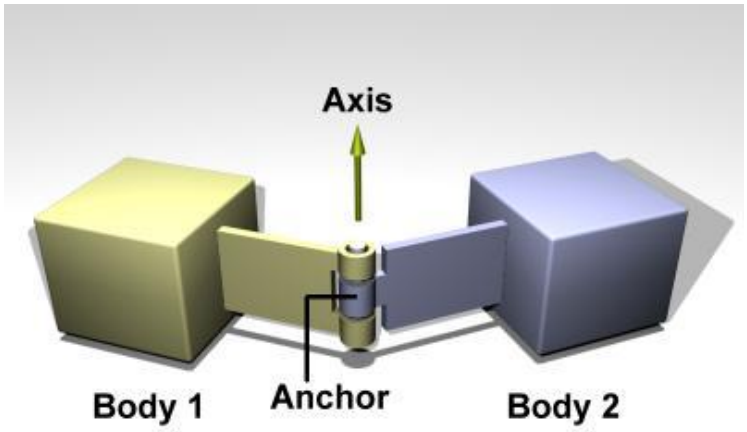


Figura 1 – Imagem ilustrativa do principio de funcionamento de uma *joint* do tipo *hinge*

2.2. Simulação de Colisões

A simulação de colisões é feita separadamente da simulação da dinâmica, e ocupa-se da forma dos corpos. A sua função é exclusivamente a de determinar os pontos de contacto entre objectos. Estes pontos são usados para criar junções especiais, *contact joints*, que são normalmente usadas para evitar que os objectos penetrem uns nos outros. Estas junções permitem também simular fricção entre os pontos de contacto, aplicando-se forças na direcção do atrito, perpendiculares à normal das superfícies, como representado na figura 8.

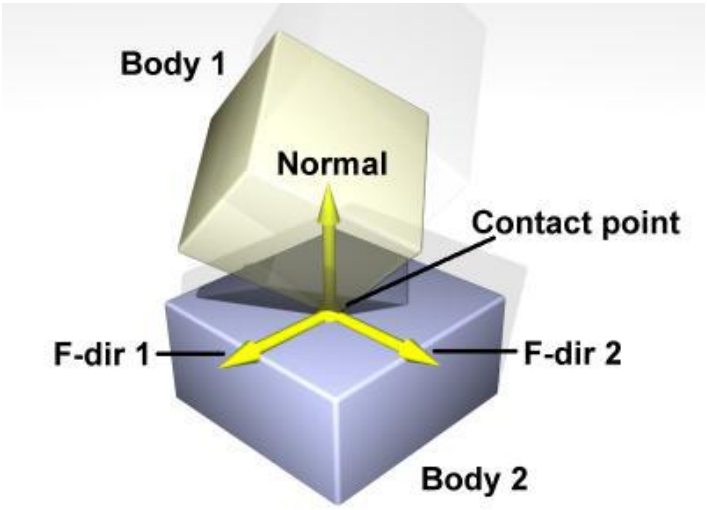


Figura 2 – Imagem ilustrativa do principio de funcionamento de uma *joint* de contacto

3. Configuração do simulador

A configuração do simulador e das entidades que povoam o *mundo virtual*[2], é realizada por intermédio de um conjunto de ficheiros *XML* [3] que são lidos no instante de arranque da aplicação. O *SimTwo* permite desta forma a definição de estruturas hierárquicas de informação que contempla no nível mais elevado desta hierarquia o ficheiro *XML, scene.xml*. A partir deste nível são introduzidas as classes ou secções, que definem, a configuração do robô, os obstáculos, os objectos e a pista, sendo que esta última expressa o ambiente das trajectórias definidas para o robô.

A principal diferença entre obstáculos e objectos é que os primeiros representam estruturas inamovíveis enquanto que os objectos são elementos livres com dinâmica.

Na tabela 1 apresenta-se um resumo das classes abrangidas pelo ficheiro ***scene.xml***.

Tabela 1 – Classes abrangidas pela estrutura hierarquia mais elevada do *SimTwo*

Classes	Descrição
Robot	Configuração da estrutura do robô
Things	Configuração de corpos dinâmicos
Obstacles	Configuração de obstáculos (inamovíveis)
Track	Configuração de ambientes com trajectórias desenhadas no pavimento

[2] Subentende-se todos os objectos constituintes do meio ambiente desenvolvido para a simulação, tais como robôs, objectos, obstáculos e pista.

Das várias classes descritas anteriormente, a classe *robot* é a única instância controlável na plataforma de simulação. A configuração desta classe é estabelecida recorrendo a estruturas virtuais que são definidas por objectos. Estes podem ser divididos em dois grupos, um que incorpora os objectos básicos, *solids*, permitindo definir a estrutura física do robô, e outro que incluí os objectos mais complexos, tais como sensores, articulações, que permitem a interacção com o ambiente. Os objectos básicos possuem algumas características fundamentais, tais como forma, massa, dimensão, permitindo simular de uma forma realista a dinâmica dos corpos rígidos. Na tabela 2 exibe-se os diferentes tipos de objectos básicos suportados pelo *SimTwo* e as suas propriedades. Os restantes objectos são apresentados na tabela 3. Um exemplo de código desta implementação é mostrado de seguida:

```
<solids>
    <belt>
      <ID value='1'/>
      <mass value='20'/>
      <size x='1.5' y='0.6' z='0.02'/>
      <pos x='0' y='0' z='0.1'/>
      <rot_deg x='0' y='0' z='0'/>
      <color_rgb r='0' g='200' b='0'/>
    </belt>
```

Tabela 2 – Objectos básicos do *SimTwo* e suas propriedades

Objecto	Descrição	Propriedade	Componente	Descrição da Propriedade
Cuboid	Paralelepípedo	Id	value	Identificação do objecto
		Mass	value	Massa do objecto, em Kg
Cylinder	Cilindro	Size	x, y, z	Tamanho, em metros
Belt	Tapete rolante	Pos	x, y ,z	Posição no referencial mundo
Sphere	Esfera	Rot_deg	x, y, z	Rotação, em graus
		Color_rgb	r, g, b	Cor na base RGB (8bits)

Tabela 3 − Objectos mais complexos suportados pelo *SimTwo*

Objecto	Descrição
Sensors	Sensores
Articulations	Articulações
Wheels	Rodas
Shells	Superfícies de contacto

A utilização dos objectos descritos na tabela 3 confere um nível de complexidade mais elevado à configuração dos robôs. As *shells* possuem a finalidade de tornar a forma do robô mais complexa e de resolver a limitação imposta pela biblioteca *ODE*, pois esta não implementa colisões entre cilindros. As *wheels* e as *articulations* correspondem ao sistema de actuação dos robôs, permitindo definir eixos articulados entre corpos. Por fim, os *sensors* permitem introduzir sensores no robô, embora presentemente só se encontram disponíveis sensores que medem distâncias (simlares aos sensores de infravermelhos), sensores capacitivos, sensores indutivos e sensores de linha.

Na tabela 4 apresentam-se as principais propriedades existentes nos objectos mais complexos.

Tabela 4 − Propriedades dos objectos mais complexos abrangidos pelo *SimTwo*

Propriedade	Componente	Objecto	Descrição
Motor	r_i, k_i, v_{max}, i_{max}, active	*Articulations*/*Wheels*	Parâmetros do motor
Gear	ratio	*Articulations*/*Wheels*	Relação da caixa redutora
Friction	B_v, f_c, coulomblimit	*Articulations*/*Wheels*	Parâmetros da fricção nos actuadores
Encoder	ppr, mean,	*Articulations*/*Wheels*	Parâmetros do codificador

	stdev		óptico
Controller	mode, k_p, k_i, k_d, k_f, active	*Articulations/Wheels*	Parâmetros do controlador
Axis	x, y, z	*Articulations/Wheels*	Parâmetros do eixo da roda
type	value	*Articulations*	Tipo de articulação
Connect	B1, B2	*Articulations*	Ligação entre dois objectos
Pos	x, y, z	*Articulations/Sensors /Sheels*	Posição no referencial mundo
Limits	Min, Max	*Articulations*	Limites da articulação
Beam	length, initial_width, final_width	*Sensors*	Parâmetros dos sensores
Rot_deg	x, y, z	*Shells/Sensors*	Rotação, em graus
Color_rgb	r, g, b	*Shells/Sensors*	Cor na base RGB (8bits)
Omni	—	*Wheels*	Define a roda como omnidireccional
tyre	Mass, radius, width, centerdist	*Wheels*	Parâmetros da roda

4. Modos de Controlo

O *SimTwo* oferece dois níveis de controlo do robô. Um sistema de controlo de baixo nível responsável pelo controlo dos motores que, por defeito, efectua 100 ciclos por segundo (invocado periodicamente de 10 ms em 10 ms) e um controlador de mais alto nível implementado pelo utilizador, que permite desenvolver tarefas de controlo mais complexas e que, por defeito, corre com um período de 40 ms (25 ciclos por segundo). Este controlador pode ser implementado recorrendo a um programa remoto que comunica com o *SimTwo* através de protocolo *UDP* ou porta série, ou recorrendo a uma linguagem de *script* que se baseia na biblioteca *Pascal Script* [4] editável e compilável no próprio simulador. É de salientar que o controlador de baixo nível encontra-se desenvolvido de origem, mas permite as alterações de muitos parâmetros.

4.1. Controlo dos robôs via *Script*

Como foi referido anteriormente, uma das opções para controlar os robôs é através de um programa que é escrito no próprio *SimTwo*. Esta opção tem a vantagem de não precisar de um ambiente externo de desenvolvimento.

Para editar o *script*, o *SimTwo* fornece um mini-IDE com algumas facilidades de edição em relação a um editor genérico.

Em anexo podem ser encontrados alguns tipos e funções disponíveis para utilização no *script*.

4.2. Controlo dos robôs via Rede

Uma outra opção para a implementação de um controlador é usar o mecanismo de comunicação do *SimTwo* via *UDP*. Quando esta opção está seleccionada, a cada instante de controlo o SimTwo envia um pacote *UDP*, com, por exemplo, o estado do robô e dos seus sensores. Uma aplicação externa pode receber essa informação, implementar o seu controlador, e responder com outro pacote *UDP* onde são especificados os sinais de controlo.

O porto de ligação via *UDP* pode ser configurado na janela de configuração no *tab I/O* e recorrendo às funções:

```
procedure ReadUDPData(): String;
function WriteUDPData(ToIP: String; ToPort: LongInt; s: String);
```

4.3. Controlo dos robôs por Porta Série

Ainda outra opção para a implementação de um controlador é usar o mecanismo de comunicação do SimTwo via Porta Série. A partir do script podem ser enviados e recebidos dados via uma port série. O próprio script deverá utilizar esses dados para gerar os sinais de controlo, possibilitando assim que uma aplicação externa implemente o seu controlador. A ligação série pode ser configurada no *tab* I/O. Pode-se ler e escrever na porta série recorrendo às funções:

```
procedure ReadComPort(): String;
function WriteComPort(s: String);
```

Referências

[1] Costa, P. (2011). *Simtwo*. Acedido em: http://paginas.fe.up.pt/~paco/wiki/index.php?n=Main.SimTwo.

[2] Smith, R. (2009). *Open dynamics engine*. Acedido em: http://www.ode.org/.

[3] W3C-WorldWideWeb Consortium (2009). *Extensible markup language (xml)*. Acedido em: http://www.w3.org/XML/.

[4] Kok, C. (2009). *Pascal script*. Acedido em: http://wiki.freepascal.org/Pascal_Script.

Anexo

Tipos disponíveis no Script

```
type
  TAxisPoint = record
    pos: double;
    speed: double;
    final_time: double;
  end;

  TAxisState = record
    pos: double;
    vel: double;
  end;

  TState2D = record
    x: double;
    y: double;
    angle: double;
  end;

  TPoint3D = record
    x: double;
    y: double;
    z: double;
  end;
```

Funções disponíveis no Script

```
procedure ExceptionType(): TIFException;
procedure FloatToStr(e: Extended): String;
procedure GetArrayLength(arr): LongInt;
procedure GetAxisEnergy(R: LongInt; i: LongInt): Double;
procedure GetAxisI(R: LongInt; i: LongInt): Double;
procedure GetAxisIndex(R: LongInt; ID: String; i: LongInt): LongInt;
procedure GetAxisOdo(R: LongInt; i: LongInt): LongInt;
procedure GetAxisPos(R: LongInt; i: LongInt): Double;
procedure GetAxisPosDeg(R: LongInt; i: LongInt): Double;
procedure GetAxisPosRef(R: LongInt; i: LongInt): Double;
procedure GetAxisPosRefDeg(R: LongInt; i: LongInt): Double;
procedure GetAxisSpeed(R: LongInt; i: LongInt): Double;
procedure GetAxisSpeedDeg(R: LongInt; i: LongInt): Double;
procedure GetAxisSpeedRef(R: LongInt; i: LongInt): Double;
procedure GetAxisSpeedRefDeg(R: LongInt; i: LongInt): Double;
procedure GetAxisState(R: LongInt; i: LongInt): TAxisState;
procedure GetAxisStateRef(R: LongInt; i: LongInt): TAxisState;
procedure GetAxisTorque(R: LongInt; i: LongInt): Double;
procedure GetAxisTrajPoint(R: LongInt; i: LongInt; idx: LongInt): TAxisPoint;
procedure GetAxisTWPower(R: LongInt; i: LongInt): Double;
procedure GetAxisU(R: LongInt; i: LongInt): Double;
procedure GetAxisUIPower(R: LongInt; i: LongInt): Double;
procedure GetAxisWayPoint(R: LongInt; i: LongInt; idx: LongInt): TAxisPoint;
procedure GetBeltSpeed(R: LongInt; i: LongInt): Double;
procedure GetFrictionDef(R: LongInt; i: LongInt): TFrictionDef;
procedure GetMotorControllerPars(R: LongInt; i: LongInt):
TMotorControllerPars;
procedure GetRobotPos2D(R: LongInt): TState2D;
procedure GetRobotTheta(R: LongInt): Double;
procedure GetRobotVel2D(R: LongInt): TState2D;
procedure GetRobotVx(R: LongInt): Double;
```

```
procedure GetRobotVy(R: LongInt): Double;
procedure GetRobotW(R: LongInt): Double;
procedure GetRobotX(R: LongInt): Double;
procedure GetRobotY(R: LongInt): Double;
procedure GetSensorVal(R: LongInt; i: LongInt): Double;
procedure GetSolidCenterOfMass(R: LongInt; i: LongInt): TPoint3D;
procedure GetSolidColor(R: LongInt; i: LongInt): TRGBAColor;
procedure GetSolidIndex(R: LongInt; ID: String): LongInt;
procedure GetSolidLinearVel(R: LongInt; i: LongInt): TPoint3D;
procedure GetSolidMass(R: LongInt; i: LongInt): Double;
procedure GetSolidPos(R: LongInt; i: LongInt): TPoint3D;
procedure GetSolidTheta(R: LongInt; i: LongInt): Double;
procedure GetSolidVx(R: LongInt; i: LongInt): Double;
procedure GetSolidVy(R: LongInt; i: LongInt): Double;
procedure GetSolidVz(R: LongInt; i: LongInt): Double;
procedure GetSolidX(R: LongInt; i: LongInt): Double;
procedure GetSolidY(R: LongInt; i: LongInt): Double;
procedure GetSolidZ(R: LongInt; i: LongInt): Double;
procedure GetThingColor(T: LongInt; c: LongInt): TRGBAColor;
procedure High(x): Int64;
procedure IDispatchInvoke(Self: IDispatch; PropertySet: Boolean; Name:
String; Par: !OPENARRAYOFVARIANT): Variant;
function  Inc(x);
function  Insert(s: AnyString; s2: AnyString; iPos: LongInt);
procedure Int(e: Extended): Extended;
procedure Int64ToStr(i: Int64): String;
procedure IntToStr(i: Int64): String;
procedure IsMotorActive(R: LongInt; i: LongInt): Boolean;
procedure KeyPressed(k: LongInt): Boolean;
procedure Length(s): LongInt;
function  LoadJointWayPoints(r: LongInt; JointPointsFileName: String);
procedure Low(x): Int64;
procedure Lowercase(s: AnyString): AnyString;
```

```
 procedure NormalizeAngle(ang: Double): Double;
 procedure Null(): Variant;
 procedure Padl(s: AnyString; l: LongInt): AnyString;
 procedure Padr(s: AnyString; l: LongInt): AnyString;
 procedure Padz(s: AnyString; l: LongInt): AnyString;
 procedure Pi(): Extended;
 procedure Pos(SubStr: AnyString; S: AnyString): LongInt;
 procedure Rad(angle: Double): Double;
 function  RaiseException(Ex: TIFException; Param: String);
 function  RaiseLastException();
 procedure ReadComPort(): String;
 procedure ReadUDPData(): String;
 procedure Replicate(c: Char; l: LongInt): String;
 function  ResetAxisEnergy(R: LongInt; i: LongInt);
 procedure RotateAndTranslate(rx: Double; ry: Double; px: Double; py:
Double; tx: Double; ty: Double; teta: Double): Double;
 procedure RotateAroundPoint(rx: Double; ry: Double; px: Double; py: Double;
cx: Double; cy: Double; teta: Double): Double;
 procedure Round(e: Extended): LongInt;
 procedure Sat(a: Double; limit: Double): Double;
 function  SaveJointWayPoints(r: LongInt; JointPointsFileName: String);
 function  SetArrayLength(arr count: LongInt);
 function  SetAxisPosRef(R: LongInt; i: LongInt; aPos: Double);
 function  SetAxisSpeedRef(R: LongInt; i: LongInt; aSpeed: Double);
 function  SetAxisSpring(R: LongInt; i: LongInt; k: Double; ZeroPos: Double);
 function  SetAxisStateRef(R: LongInt; i: LongInt; aState: TAxisState);
 function  SetAxisTrajPoint(R: LongInt; i: LongInt; idx: LongInt; LP:
TAxisPoint);
 function  SetAxisVoltageRef(R: LongInt; i: LongInt; aVoltage: Double);
 function  SetBeltSpeed(R: LongInt; i: LongInt; nSpeed: Double);
 function  SetFrictionDef(R: LongInt; i: LongInt; nBv: Double; nFc: Double;
nCoulombLimit: Double);
 function  SetLength(s NewLength: LongInt);
```

```
 function  SetMotorActive(R: LongInt; i: LongInt; nState: Boolean);
 function  SetMotorControllerPars(R: LongInt; i: LongInt; nKi: Double; nKd:
Double; nKp: Double; nKf: Double);
 function  SetRobotPos(R: LongInt; x: Double; y: Double; z: Double; teta:
Double);
 function  SetSolidColor(R: LongInt; I: LongInt; Red: Byte; Green: Byte; Blue:
Byte);
 function  SetThingColor(T: LongInt; c: LongInt; Red: Byte; Green: Byte; Blue:
Byte);
 procedure Sign(a: Double): Double;
 procedure Sin(e: Extended): Extended;
 procedure SizeOf(Data): LongInt;
 procedure Sqrt(e: Extended): Extended;
 procedure StrGet(S: String; I: LongInt): Char;
 procedure StrGet2(S: String; I: LongInt): Char;
 procedure StringOfChar(c: Char; I: LongInt): String;
 function  StrSet(c: Char; I: LongInt; s: String);
 procedure StrToFloat(s: String): Extended;
 procedure StrToInt(s: String): LongInt;
 procedure StrToInt64(s: String): Int64;
 procedure StrToIntDef(s: String; def: LongInt): LongInt;
 procedure TranslateAndRotate(rx: Double; ry: Double; px: Double; py:
Double; tx: Double; ty: Double; teta: Double): Double;
 procedure Trim(s: AnyString): AnyString;
 procedure Trunc(e: Extended): LongInt;
 procedure Unassigned(): Variant;
 procedure Uppercase(s: AnyString): AnyString;
 procedure VarIsEmpty(V: Variant): Boolean;
 procedure VarIsNull(V: Variant): Boolean;
 procedure VarType(V: Variant): Word;
 function  WriteComPort(s: String);
 function  WriteLn(S: String);
 function  WriteUDPData(ToIP: String; ToPort: LongInt; s: String);
```

```
procedure WStrGet(S: WideString; I: LongInt): WideChar;
function  WStrSet(c: WideChar; I: LongInt; s: WideString);
```